Sinfonia: A New Paradigm for Building Scalable Distributed Systems

Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, Christos Karamanolis
HP Labs and VMware

Motivation

- Data Center applications
- Basic Underlying services
  - Clustered file systems, lock managers, group communication services…
- Distributed State
  - Consistency, replication, failures
- Existing Solutions:
  - Message-passing protocols: replication, cache consistency, group membership, file data/metadata management
  - Databases: powerful but expensive/inefficient
  - Distributed Shared Memory – need to meet additional requirements (ACID)

Background

- Wait-free mutual exclusion
- Incr(x) … lock()… x++; unlock()
- TST
- LL, SC
- C&S
- Mini transaction

Wait-free

Lock()
read(y,x)
Y++; ----------→
Write(y,x) write y into x
Ulock()

Critical section could become blocking if lock holder blocks

Instead use atomic instructions. Say the CPU supported atomic_incr(x);
No interrupts during the execution on the instruction
Can implement increment operation in a wait-free manner

Wait-free using TST

- TST (testandset)
- Tst(x,true) {temp=x;
- set x = true;
  return temp}

While tst(x);
Do something
X=false

5

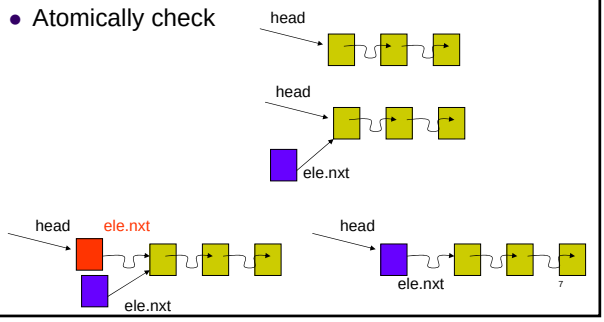
MIPS architecture

- Load linked, store conditional
- LL(register,location) --- bit is set for the location
- SC(location, value) - store if bit is still set else fail
- SC returns 1 if successful or 0 otherwise
- If interrupt or any other write on location, bit is reset
- Entry: LL(reg,location)
- if reg==1 go to entry
- if SC(1,location) ==0 go to Entry
- ….. Critical section
- Store(0,location) ……. unlock
-

6

Compare and swap (CAS)

- C&S(x,v1,v2)
- Atomically check



7

Adding an element using CAS

- CAS(head,ele.nxt,ele)
- Return true if successful else return false

8

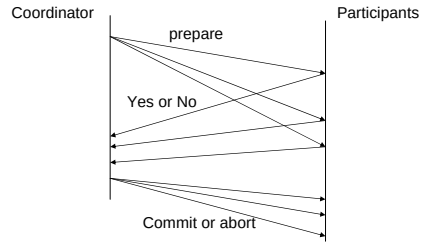
Back ground (Commit Protocols)

- Atomically update a state on one node
  - -- lock, wait-free etc
- Atomically update on N nodes
- Need agreement on action at N independent sites
- In spite of network or node failures
- Solution: Commit Protocols

9

2 Phase commit

- Prepare Phase, Commit phase



If all vote YES, send commit else send abort

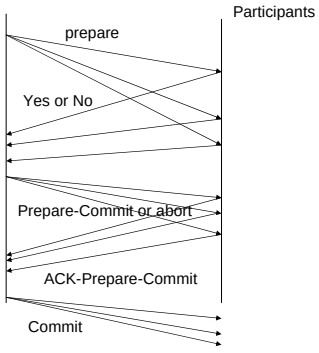
10

2 phase commit

- Participant failure
  - Coordinator can unilaterally abort
- Coordinator failure
- Participant: If Commit or abort received, then take appropriate action
- If in Prepare state and voted NO
  - Can abort unilaterally
- If in Prepare state and Voted YES
  - Limbo (need to hear from Coordinator)
  - Blocking

11

3 phase commit



12

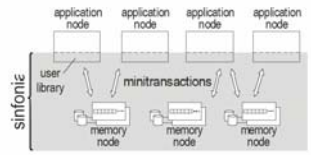
3PC (non-blocking)

- Add an additional phase prepare-commit
- After prepare, if everyone votes send prepare-to-commit
- If everyone, ACKS prepare-to-commit, then send COMMIT
- If anyone sees prepare-to-commit
  - Knows everyone else voted YES
- Can take unilateral decision on Commit
- After failure, COMMIT iff any up node has seen prepare-to-commit message

13

Sinfonia

- Provides a lightweight minitransaction primitive
- Provides Distributed Shared Memory as a service
- Atomically update nodes in a distributed system
- ACID guarantees



14

Sinfonia

- Use of CAS in minitransaction, allows batched atomic updates
  - No separate network operations per operaion
- Reduced
- In DB, updates and commit protocol are separate
- In sinfonia, minitransaction updates are piggybacked along with commit message

15

Assumptions

- Assumptions: Datacenter environment
  - Trustworthy applications
  - Stable storage available
  - Low, steady latencies --- clusters,racks
  - No network partitions
- Goal: help build distributed infrastructure applications
  - Fault-tolerance, Scalability, Consistency and Performance

16

Design Principles

Principle 1: Reduce operation coupling to obtain scalability. [linear partitioned address space]

Partitioned address space: *(mem-node-id, addr)*

- Divided among nodes
- Exposing node-id can exploit locality

Principle 2: Make components reliable before scaling them. [fault-tolerant memory nodes]

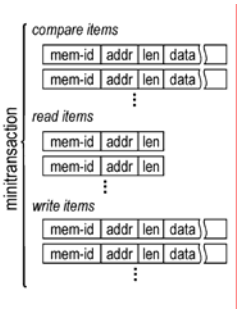
17

Minitransactions

- Consist of:
  - Set of compare items
  - Set of read items
  - Set of write items
- Semantics:
  - Check data in compare items (equality)
  - If all match, then:
    - Retrieve data in read items
    - Write data in write items
  - Else abort

18

Mini-transaction details



- Mini-transaction
  - Check compare items
  - If match, retrieve data in read items, modify data in write items
- Example:

```
t = new Minitransaction()
t->cmp(node-X:10, 4, hello)
t->cmp(node-Y:12, 4, world)
t->write(node-X:10, 4, world)
t->write(node-Y:12, 4, hello)
Status = t->exec_and_commit()
```

19

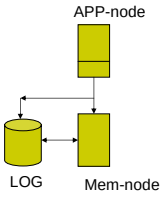
Examples of Minitransactions

- Examples:
  - Swap
  - Compare-and-Swap
  - Atomic read of many data
  - Acquire a lease
  - Acquire multiple leases
  - Change data if lease is held
- Use Minitransaction to implement Consistency in Distributed Systems

20

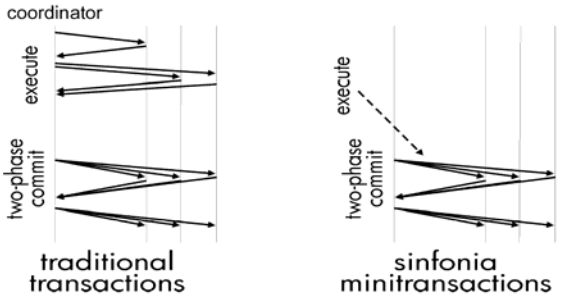
Mini transaction Protocol

- Use Minitxn + 2 PC commit
- Did not want to use 3PC
- Handle blocking problem
- Recovery coordinator
- Optimized 2pc
- Locking → isolation
- Redo log



21

Minitransactions



22

Redo Log

- Multiple data structures

Name	Description	On stable storage
redo-log	minitransaction redo-log	yes, sync
in-doubt	*tid*'s not yet committed or aborted	no
forced-abort	*tid*'s forced to abort (by recovery)	yes, sync
decided	*tid*'s in redo-log with outcome	yes, async
all-log-tids	*tid*'s in redo-log	no

- Memory node recovery using redo log
- Log garbage collection
  - Garbage collect only when transaction has been durably applied at every memory node involved

23

Recovery from coordinator crashes

- Recovery Coordinator periodically probes memory node logs for orphan transactions
- Phase 1: requests participants to vote 'abort'; participants reply with previous existing votes, or vote 'abort'
- Phase 2: tells participants to commit i.f.f all votes are 'commit'
- Recovery coordinator periodically probes for tids ad knows the node-ids of tids that are in-doubt
- Recovery coordinator cleans up
  - Ask all participants for existing vote (or vote "no" if not voted yet)
  - Commit iff all vote "yes"

24

Recovery from participant crashes

- Block until participant recovery
- No recovery coordinator for mem node crashes
- Lose stable storage – recover from a backup
- keep stable storage – recover using redo
  - Processed-point variable (check point)
  - Redo-log & decided list
  - For some minitransaction tx, (not in decided list but in redo log) contact the set D of memory nodes of tid tx

25

Log garbage collection

- Redo-log – every memory node applied *tid* to image disk

  along with *all-log-tids* list, *in-doubt* list & *decided* list
- Forced-abort list

  use epoch number to decide *tid* in forced-abort list to be garbage collected or not

26

Sinfonia Cluster FS

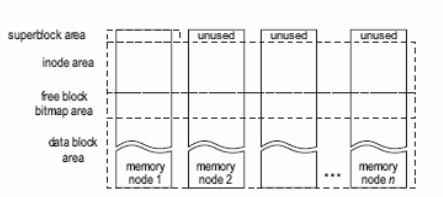


Figure 6: Data layout for SinfoniaFS. These data structures are on disk, as Sinfonia runs in mode LOG *or* LOG-REPL.

27

Sinfonia FS

- Cluster File System:
  - Cluster nodes (application nodes) share a common file system stored across memory nodes
- Sinfonia simplifies design:
  - Cluster nodes do not need to be aware of each other
  - Do not need logging to recover from crashes
  - Do not need to maintain caches at remote nodes
  - Can leverage write-ahead log for better performance
- Exports NFS v2: all NFS operations are implemented by minitxns

28

Consistent metadata update

- Use single minitxn for each NFS function

```
1 setattr(ino_t inodeNumber, sattr_t newAttributes){
2 do {
3   inode = get(inodeNumber);        // get inode from inode cache
4   newiversion = inode->iversion+1;
5   t = new Minitransaction;
6   t->cmp(MEMNODE(inode), ADDR_IVERSION(inode),
      LEN_IVERSION, &inode->iversion); // check inode iversion
7   t->write(MEMNODE(inode), ADDR_INODE(inode),
      LEN_INODE, &newAttributes); // update attributes
8   t->write(MEMNODE(inode), ADDR_IVERSION(inode),
      LEN_IVERSION, &newiversion);  // bump iversion
9   status = t->exec_and_commit();
10  if (status == fail) ... // reload inodeNumber into cache
11 } while (status == fail); }
```

29

Group Communication

- Orderred BCAST: total ordering of message

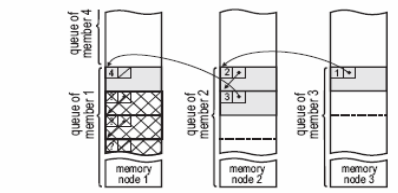


Figure 10: Basic design of SinfoniaGCS. Gray messages were successfully broadcast: they are threaded into the global list. Cross-hatched messages are waiting to be threaded into the global list, but they are threaded locally.

30

Sinfonia GCS

- Design 1: Global queue of messages
  - Write: find tail, add to it
  - Inefficient – retries require message resend
- Better design: Global queue of pointers
  - Actual messages stored in per-member queues
  - Write: add msg to data queue, use minitransaction to add pointer to global queue
- Essentially provides totally ordered broadcast

31

Evaluation – Sinfonia Service

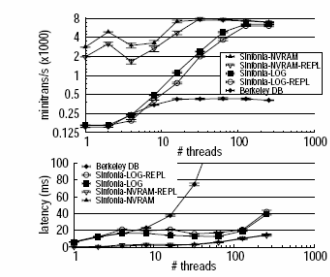


Figure 12: Performance of Sinfonia with 1 memory node.

32

Evaluation: with optimizations

- Non-batched items: standard transactions
- Batched items: Batch actions + 2PC
- 2PC combined: Sinfonia multi-site minitransaction
- 1PC combined: Sinfonia single-site minitransaction

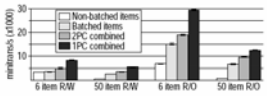
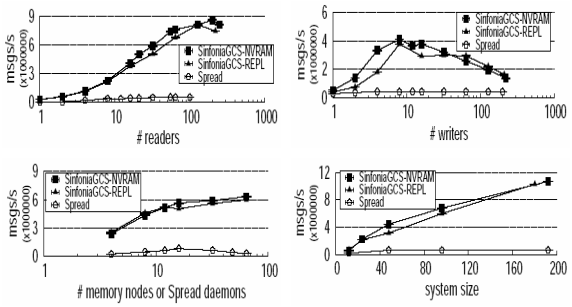


Figure 13: Performance with various combinations of techniques.

Evaluation: SinfoniaGCS



Sinfonia development effort

software engineering metrics

	sinfoniaFS	linuxNFS	sinfoniaGCS	spread toolkit
lines of code (language)	3,855 (C++)	5,900 (C)	2,492 (C++)	22,148 (C)
develop time	1 month	unknown	2 months	years
major versions	1	2	1	4

Conclusion

- Minitransactions
  - A new concept
  - ACID model
  - Shown to be powerful (all NFS operations, for example)
- Designed for Cloud services
  - Dedicated memory nodes
  - Dedicated backup 2PC coordinator
  - Not sure why no dedicated mem-node coordinator?